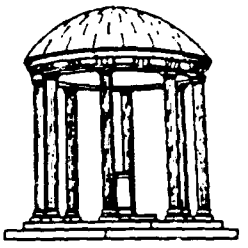
(2)

Solving Computer Graphics Problems through Boolean Combinations of Polygons

TR89-031

August, 1989

John M. Airey



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

013

Solving Computer Graphics Problems through Boolean Combinations of Polygons

John M. Airey
UNC @ Chapel Hill

0. Abstract

Several important computer graphics problems can be solved by formulating the problem as a boolean combination of sets of coplanar polygons. We describe an implementation of an efficient plane sweep algorithm, which solves these problems by triangulating the polygonal regions defined by boolean combinations of sets of polygons. Example applications include, but are not limited to, triangulating a concave polygon with holes, computing complex clipping operations, detecting modelling errors, transforming a tiling into a planar subdivision and even the hidden surface problem. We present new techniques to handle the difficulties encountered with real world input, which are typically omitted in presentations of geometric algorithms in the computational geometry literature. Finally, we describe how the above applications may be cast as boolean combinations and solved with our algorithm.

1. Introduction

Technology transfer from computational geometry to computer graphics has been slow. The published algorithms are efficient and can be proven correct, but the programmer attempting to implement these algorithms faces many obstacles. The focus of the computational geometry literature is primarily theoretical and secondarily pragmatic while the emphasis is reversed for most graphics programmers. The main difficulty faced by the practicing programmer is the liberal assumptions made by many algorithms about the input. In particular, published geometric algorithms often assume that the input is in *general position*, i.e., that no two input vertices coincide, that no two vertices have the same x-coordinate or that edges intersect only at end points, etc. This makes the analysis of the algorithm much simpler but makes the task of implementation much harder. Input that is not in general position is termed *degenerate* even though it may occur more often in practice than input in general position. Secondary difficulties are those of data representation. The universal unit of information for computer graphics is the oriented convex polygon represented by a list of vertices. Algorithms in the literature may assume some other representation for input or output.

We have developed an algorithm that handles degenerate input while maintaining good performance. Simply stated, it computes boolean operations on sets of polygons. The input may consist of such normally troublesome entities as concave polygons with holes. The output is a triangular subdivision of the plane over which the boolean function is true. Although this appears to be rather abstract, many problems encounted by graphics programmers can be cast as boolean operations on sets of polygons. We list some applications here briefly and discuss them in greater detail in section 6.

1) Many graphics algorithms demand convex polygons. Concave polygons, possibly with holes, must be decomposed into convex parts. Obviously, triangles are convex, so triangulation yields a convex decomposition. This is the simplest application of our algorithm but may be the most useful. In section 2 we discuss less obvious advantages of the triangular subdivision, such as how to join triangles together with a single pass over the triangles to get a good convex decomposition of a concave polygonal region.

2) Generalized clipping operations appear often in graphics applications. For example, CSG algorithms which operate on boundary representation data need to special case coincident coplanar faces. If two cubes are placed so that they abut and we take the union of the cubes, it is necessary to remove the portion of the faces that touch since those portions are no longer boundaries; they are inside the union of the cubes. Computing the exclusive or of those faces solves the problem. We present other instances of generalized clipping in section 6.

3) Detecting and counting overlapping polygons is another common problem. In some modeling applications, it is necessary to detect overlapping or *coincident* faces because we consider them a modelling error. A wall in a building should not be modelled with overlapping polygons because this will produce rendering errors. Detecting regions where two polygons overlap can be formulated as a boolean expression problem and computed by our algorithm. The algorithm can also be modified to count overlaps.

4) In other applications, it is necessary to cover a curved surface with a planar subdivision rather than a simpler tiling to ensure that cracks do not appear. Analogously, if a flat surface is shaded we can think of the intensity values as a curved surface. With a tiling, the cracks in the intensity surface appear as shading discontinuities. Since our program produces a triangular subdivision, it can be used to eliminate cracks and shading discontinuities.

5) Lastly, it is possible to the solve the hidden surface problem with an algorithm based on polygon clipping. Since polygon clipping can be formulated as a boolean expression it is possible to base a hidden surface program on our plane sweep boolean expression algorithm.

Our algorithm is a generalization of a O(n logn) plane sweep algorithm to triangulate a simple polygon [Mehlhorn]. Plane sweep algorithms without the triangulation feature have been used by the VLSI community to analyse circuits through boolean combinations of mask artwork [Syzmanski], [Ottman], [Niev]. The plane sweep paradigm offers several advantages which make it possible to run the algorithm on huge VLSI circuit designs with only a moderately sized machine. It is iterative rather than recursive so external devices may be used for very large input sets and the amount of primary memory required is usually less than the square root of input size.

However, VLSI applications can make assumptions about the input such as restricting edges to lie horizontally or vertically. Our presentation is the first we know of which addresses the difficult problems raised by the unrestricted input encountered in graphics applications. We offer a new interpretation of the local geometry of the transition vertex which eliminates the extensive special casing that would otherwise be necessary to handle real world input.

The only other boolean expression algorithm we know of is Weiler's algorithm [Weiler80] which ignores the geometric structure and looks only at the topologic or graph theory aspects of the problem. Ignoring the geometric structure makes it impossible to compute polygon edge intersections efficiently. If the input has 1 million vertices, Weiler's $O(n^2)$ algorithm will run about 30,000 times slower than our O(nlogn) algorithm.

We begin with a short note on representation of data in section 2. Then we present a simple triangulation algorithm in section 3 and extend it in section 4 to get the main algorithm. In Section 5 we discuss implementation issues. Section 6 presents applications we have implemented ourselves and applications which might interest others.

2. Why Triangulation?.

Although there are many ways to represent polygonal regions of the plane, such as the DCEL (doubly connected edge list) represetation of a planar graph [Shamos], or boolean combinations of halfspaces [Dobkin], we chose to represent regions of the plane with a triangular subdivision. Fortunately, it is known that any polygonal region may be triangulated and in fact many different triangulations may exist for the same polygonal region. We represent a triangular subdivision simply by listing the triangles. The triangles are represented by listing the vertices in counterclockwise order. The geometric content of the edges is thus represented indirectly. We represent the topologic content of the edges with three pointers to the neighboring triangles.

The primary reason we have developed our algorithm to output triangles is that many display devices and rendering algorithms require this representation of a surface. However, a triangular subdivision has many other desirable attributes. For example, the number of triangles is proportional to the number of vertices. This means that any operation that was originally implemented in time proportional to the number of vertices, such as computing the area of the region or testing point inclusion can be implemented with the same performance by operating on

the triangles. If it is of great importance to reduce the number of convex components of a region, we can join some of the triangles together to get a good convex decomposition in linear time. This is fortunate because computing the optimal convex decomposition takes cubic time! Many other problems in computational geometry are trivial once given a triangular subdivision [O'Rourke], [Mehlhorn]. These include path-planning and two dimensional visibility calculations which are useful to robotics.

We do not require, however, that the input be a triangular subdivision. We have kept the unit of information for the input down at the level of the directed line segment. By convention, the region the line segment bounds lies to the left as we travel from the first vertex to the next. This allows concave polygons with holes to be used in the input. It does, however, preclude self-intersecting polygons since, in that case, the orientation of some edges are not defined.

3. A Plane-sweep algorithm for triangulation.

Before we consider the triangulation of an arbitrary region of the plane defined by a boolean expression over sets of polygons we review a plane-sweep algorithm [Mehlhorn] to compute a triangulation of a simple polygon, P.

The plane sweep algorithm puts the vertices of P into a priority queue, which we call the Xqueue, with the vertices ordered lexicographically from left to right and then from top to bottom. A vertex is tagged as a start, bend or end vertex depending upon whether its neighbors in P both follow it in the Xqueue, one follows and one precedes, or both precede, respectively. A typical procedural interface to such a module is:

```
xq_init( );
xq_delete_min();
xq_insert(vertex);
xq_term();
```

The Xqueue should be implemented with something that guarantees O(log n) performance for **xq_insert** and **xq_delete_min** such as a heap [Sedgewick].

The algorithm sweeps a vertical line across the plane from left to right, stepping from vertex to vertex using **xq_delete_min**. At any point in time, the sweep line defines a vertical ordering on the edges of P that it intersects. Between the edges are *regions*. The regions will be either inside or outside P and furthermore, the sweep line will intersect these *in* and *out* regions alternately. The edges and regions currently intersected by the sweep line and their vertical order are represented by a data structure which we call the Ytable. A typical procedural interface to such a module is:

```
ytbl_init();
ytbl_insert(edge);
ytbl_delete(edge);
ytbl_findabove(vertex);
ytbl_findbelow(vertex);
ytbl_pred(edge);
ytbl_succ(edge);
ytbl_term();
```

Ideally, the Ytable should be implentated with a balanced tree such as a red-black implementation of top-down 2-3-4 trees [Guibas78] to ensure O(log n) cost for each of the above operations, excluding **ytbl_init** and **ytbl_term**.

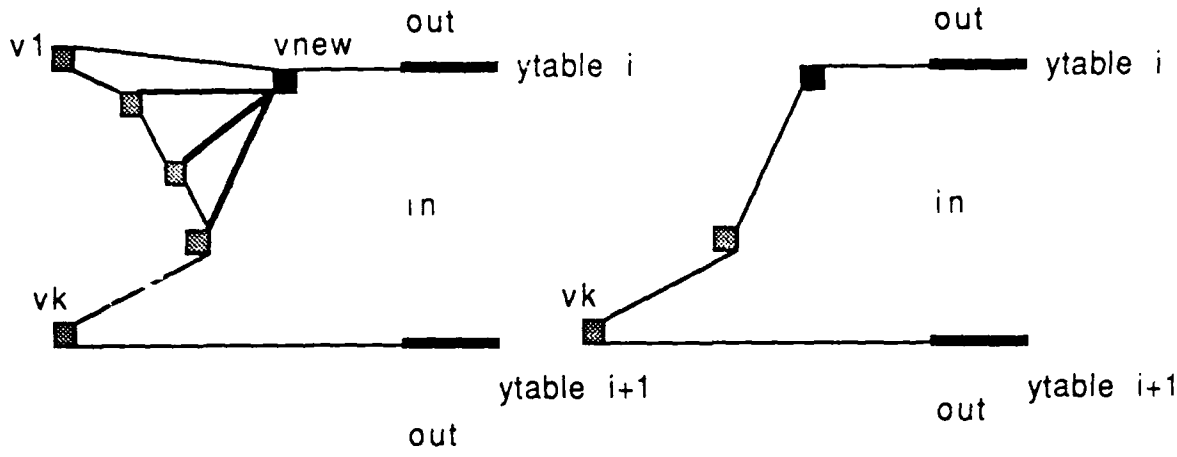
The position of the sweep line is advanced by taking a vertex from the Xqueue using **xq_delete_min** and the Ytable is updated by deleting edges that end at the current vertex and inserting edges that start at the current vertex. This reflects the changes in intersection order of edges of P with the sweep line. This maintenance of the Ytable is common to the invariant of all plane sweep algorithms. The skeleton of any plane sweep algorithm takes the form:

```
sweep()
{
    vertex v;

    xq_init( list of input polygons);
    ytbl_init( );
    while ((v = xq_delete_min()) is not null)
       transition (v);
    xq_term();
    ytbl_term();
}

transition(v)
vertex v;
{
    1. maintain the Ytable ordering invariant by deleting edges that end at v and
       inserting edges that start at v.
    2. maintain the invariant particular to this plane sweep algorithm.
}
```

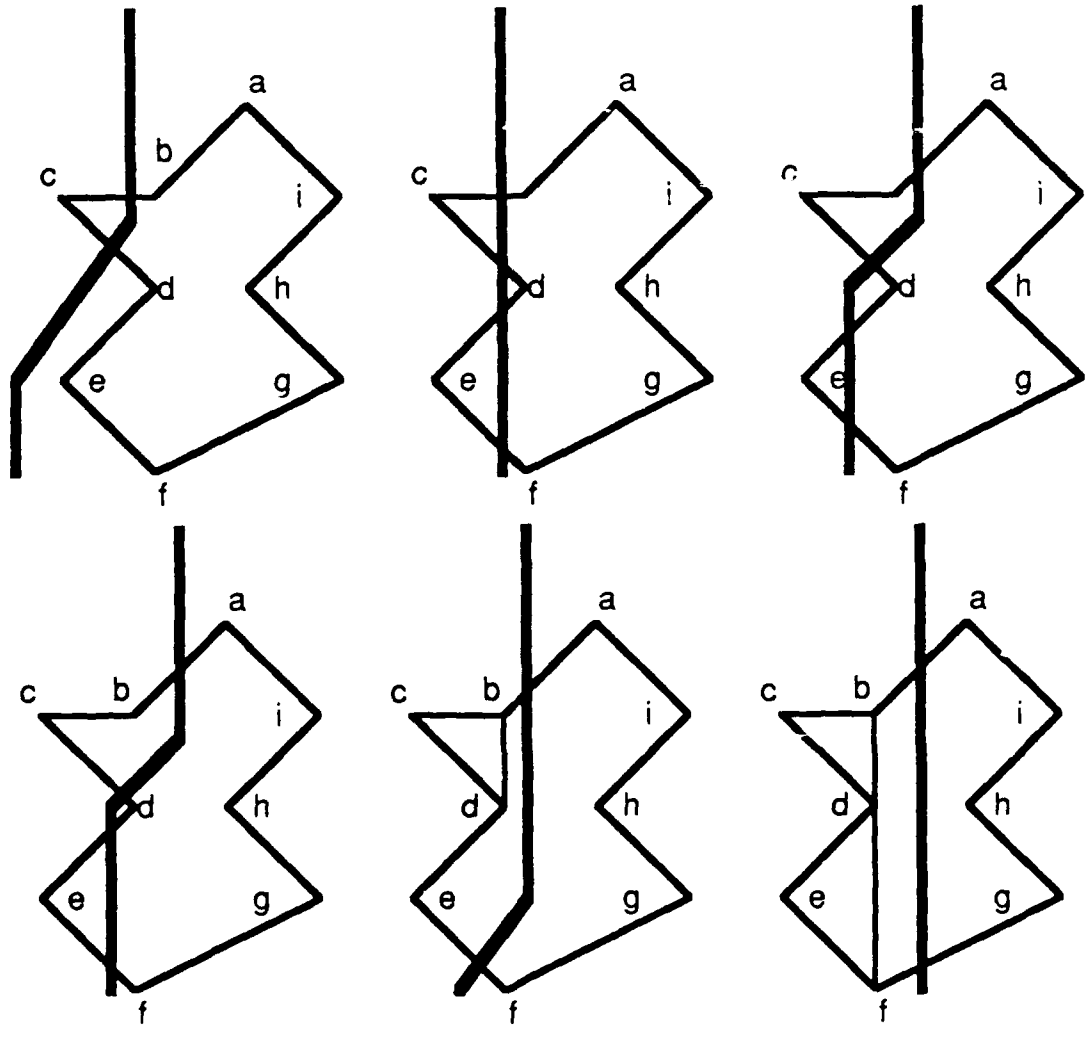
Generally a plane sweep algorithm will have some other processing at each transition. In this plane sweep algorithm we associate with every *in* region a chain of vertices, $v_1,...,v_k$ where v_1 and v_k are endpoints of the boundary edges of the region in the Ytable and edges (v_i,v_{i+1}) will become edges of the triangulation. The invariant maintained at each transition that is specific to triangulation is that no triangle can be constructed from any chain. Basically, the chain must be "concave" or have less than three vertices, i.e. if we closed the chain with an edge from v_1 to v_k we would get either a polygon that is oriented clockwise or a simple line segment.

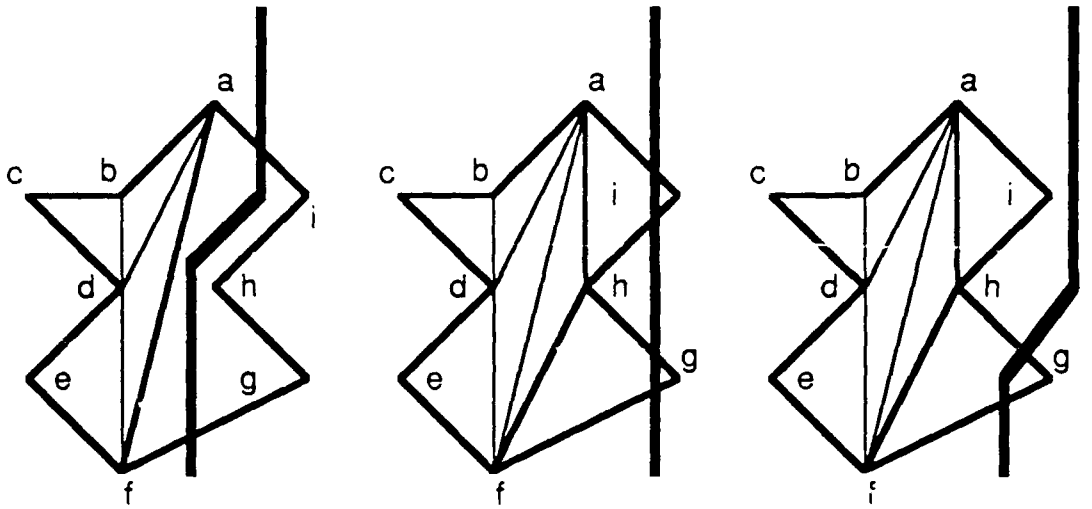
The important property of this invariant is that if we are given a chain satisfying the invariant, and a new point is added at either end of the chain, it is sufficient to check for a possible output triangle with the new point and its two closest neighbors in the chain. If no counterclockwise triangle can be constructed, then no triangle can be constructed anywhere in the chain. If a triangle can be constructed, the chain is reduced by one vertex and the process may be repeated on the reduced chain. This means that if k triangles can be constructed, O(k) steps will find them and construct them. This action of triangulating up or down a chain when given a chain and a new point is the action that maintains the invariant and produces the triangles.

The maintenance of this second invariant is depicted below. The left hand side shows a chain of five vertices associated with an *in* region and a new vertex, vnew, which has been added to the top of the chain. Since vnew,v1,v2 form a triangle, v1 is removed from the chain and the triangle is output. The process is repeated twice more with vnew, v2, v3 and vnew, v3, v4. This leaves the reduced chain seen on the right.



The algorithm must be designed to maintain this invariant and the Ytable invariant on each transition point. There are three main transition cases to consider depending upon whether the transition vertex has been tagged as a start, bend or end vertex. We consider the action of the algorithm on an example. The progress of a sweep line and the resulting triangles is depicted in the accompanying figure. The interior triangulation edges are the same thickness as the polygon if they are part of a chain and are thinner if they have been output. The sweep line appears bent in some instances because vertices with equal x-coordinate values are processed from top to bottom.





There are two possibilities for a start vertex. It can appear in an *out* region in which case the action is especially simple. The two edges that emanate from the vertex are inserted into the Ytable. They form a new *in* region and their chain is simply the new vertex. The transition on vertex *c* is an example of this case. If the point appears in an *in* region, the action is slightly more complex. The transition on vertex *h* is an example of this case. The chain that "surrounds" the transition vertex is broken into two chains and the transition vertex is appended to the end of the upper chain and the head of the lower chain. The old chain is broken at its rightmost point. Any triangles that can be constructed from these new chains are output. In this case the old chain *a,f* becomes *a,h* and *h, a, f*. The latter chain yields one triangle and is reduced to *h,f*.

If the vertex is a bend vertex, we add the new vertex to the appropriate end of the chain and then output triangles if the new addition allows us to do so. Vertex *a* is an example of this type of transition. The chain *b,d,f* becomes *a,b,d,f* and the triangles *abd* and *adf* are output and the chain becomes simply *a,f*. The entry in the Ytable which ended in *a* is replaced by the edge that starts at *a*.

As with the start vertex, there are two possibilities for an end vertex. The transition on vertex *d* illustrates the first possibility where the vertex appears in an out region. The two chains, *b,c* and *e*, are triangulated with the new vertex and then they are joined. The two edges that bounded the *out* region are deleted from the Ytable and the neighboring regions become one region. In this case one triangle was produced when the chain *b,c,d* was reduced to *b,d*. The chains *b,d* and *d,e* are then joined to become chain *b,d,e*. The second possibility, that the vertex appears in an *in* region is illustrated by vertex *i*. The chain becomes closed and is entirely reduced to triangles. In this case there is only one triangle produced. The edges that ended in vertex *i*, *ia* and *hi*, are deleted from the Ytable and the neighboring *out* regions are merged into one *out* region.

The correctness of any part of the triangulation rests upon the idea that the chains are kept "concave" and that maintaining the invariant does not result in overlapping triangles. The only questionable case is when an end point appears in an *out* region. Mehlhorn provides a rigorous proof of correctness for this case. All plane sweep algorithms have the property that maintaining the Ytable and getting the next value from the Xqueue can be performed in O(log n) time. Since there are n vertices, the result is an O(nlogn) algorithm. Maintaining the chain invariant can also be done within this bound. This follows from the fact that the number of triangles is proportional to the number of vertices and only a constant amount of work was done for each triangle.

4. Generalizing the Simple Triangulation Algorithm

We now present the extensions and generalizations of the plane-sweep triangulation algorithm necessary to triangulate an arbitrary region of the plane defined by a boolean expression over sets of polygons rather than simply the interior of one polygon.

First, we will need some mechanism to handle transition vertices formed by the intersection of polygon edges. We will also need some technique to determine what regions of the plane satisfy

the boolean function. We can't rely on alternating *in* and *out* regions as in the simple triangulation algorithm.

The last problem is the most difficult to handle in practice. If the input is not in general position, we also must expect vertices to coincide with other vertices and edges. A nasty consequence is that a transition vertex may have any number of edges entering it from behind the sweep line and any number of edges exiting it ahead of the sweep line. This means that each transition vertex can not be neatly characterized as a start, bend or end vertex. Trying to process vertices that coincide as independent events is not sufficient.

A programmer could methodically transform the explanation of what to do in each of the transition configurations in the simple polygon triangulation algorithm with one case for each of the start, bend and end transitions described in the simple triangulation algorithm above. However, now the number of configurations is no longer finite and cannot be handled with such a taxonomic approach. We propose a new interpretation for these complicated transitions which allows them to be processed without special cases.

4.1 Handling Transitions Introduced by Edge Intersections

Edge intersections must be detected so that they may be treated as transition points in the same sense as vertices of the input polygons are treated. Each time an edge is inserted into the Ytable we check whether it intersects the edges that are adjacent in the Ytable. Similarly, when an edge is deleted from the Ytable we check whether the edges that are now adjacent intersect. A little thought convinces one that this will catch all intersections. When an intersection is detected and it is ahead of the sweep line it is inserted into the Xqueue. The **xq_insert()** routine is used for this purpose. This will not change the O(logn) complexity of operations done at each transition but it may increase the number of transitions. In the worst case the number of intersections could be $O(n^2)$ but in practice it is much less than n.

4.2 Determining which Regions Satisfy the Boolean Expression

In the triangulation algorithm above we kept track of whether a region was an *in* region or an *out* region. The regions alternated and were bounded on either side by neighbors in the Ytable. This implies that each edge in the Ytable could be treated as a transition from an *in* region to an *out* region. Now we will keep track of whether a region between two edges in the Ytable is *in* or *out* with respect to our desired boolean function. It is no longer true that regions will alternate so our interpretation of edges in the Ytable as transitions from *in* regions to *out* regions or vice versa will have to be augmented the concept of a non-transition edge. If two neighboring regions are *in* regions, or *out* regions, the edge between them is a *non-transition* edge.

Since the boolean expression evaluation for a region depends upon what sets of inputs "cover" the region we associate an array with each region that has one entry for each distinct set in the input. The entry will hold a count of the number of polygons from that input set which cover the region. This array can be evaluated by a function which will determine whether the region is *in* or *out*. When an edge from a set of polygons is inserted into the Ytable, the count for that set in the Ytable region bounded by the edge is incremented. Similarly, an edge deletion requires decrementing the appropriate counter.

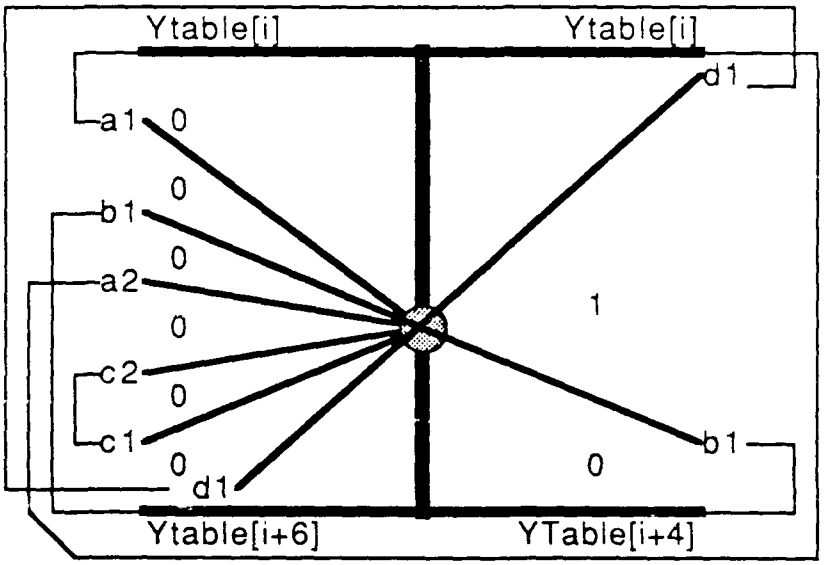
4.3 Processing Complex Transitions Caused by Degenerate Input

If we assumed that the input was in general position, the extensions outlined in sections 4.1 and 4.2 would be sufficient. The algorithm will run in time O((n+s)(v+logn)) where n is the number of vertices in the input, s is the number of vertices created by edge intersections and v is the size of the array used to evaluate the boolean expression for a region or equivalently is equal to the number of distinct variables in the boolean expression. Typically v will be a small integer so that the complexity can be simplified to O((n+s)(logn)).

However, if the program is to be useful it must be modified to handle degenerate input. When many vertices in the input coincide with other vertices or with other edges, the transition vertex may have any number of edges entering it from behind the sweep line and any number of edges

exiting after the sweep line.

The following diagram gives an example of such a situation. The scan-line is drawn vertically as a thick line, the transition vertex is the circle and the entries in the Ytable that intersect the transition vertex before and after the transition are shown as polygon edges while the entries in the Ytable above and below the transition vertex are shown as horizontal thick lines. The regions between edges are labeled with either a 1 or 0 depending upon whether the evaluation of the boolean function is true or false. The labeling of the edges indicates whether the edge extends through the vertex or ends or starts at the vertex. For example, edges a1 and a2 are sequential edges in polygon a and they both end at the transition vertex. Edges b1 and d1 intersect at the transition vertex and have been drawn over the the transition vertex to emphasize that they do not end or start at the vertex. The numbering of the edges reflects the counterclockwise orientation of the polygons. Possible global connections of these edges are suggested by the thin lines connecting edges from a common polygon.



Sets are g1={a}
g2={b,c,d}

Expression is
g1-g2

We now show the geometric interpretation of the transition vertex that allows it to be processed without special cases for degenerate situations. Our algorithm will consider each of the edges in counterclockwise order around the transition vertex. If an edge is a transition from a 0-region to a 1-region we will call it a R(ising) edge and if it is a transition from a 1-region to a 0-region we will call it a a F(alling) edge. Other edges are C(onstant) edges. In our example, *b1* is an R edge to the right of the scan-line and *d1* is an F edge to the right of the scan-line. All other edges are C edges. We will look for pairs of R and F edges as we travel counterclockwise around the transition vertex. Clearly, two R edges seperated by zero or more C edges cannot occur, similarly for two F edges. This implies that R and F edges seperated by zero or more C edges will occur in pairs. This is the necessary abstraction; we consider pairs of R and F edges instead of simply pairs of edges.

Each R-F pair may be treated just as the pair of edges bounding the "in" region were treated in the simple polygon triangulation algorithm. We need a mildly complicated loop structure to detect the R-F pairs but once they are found they may be handled with a simple case or if-elseif type control structure:

1. The R and F pair are to the left of the scan-line.
   Depending upon which edge was encountered first, this corresponds to one or the other of the end cases described in the triangulation algorithm.

2. The R edge is on one side of the scan-line, the F edge is on the other.
   This is equivalent to the bend case in the triangulation algorithm.

3. The R and F pair are to the right of the scan-line.
   Depending upon which edge was encountered first, this corresponds to one or the other of the start cases described in the triangulation algorithm.

We may now consider the details of the loop used to detect the R,F pairs of edges in the order described. The Ytable will provide the counterclockwise ordering if we find YTable[i] using **ytbl_findabove()** and use **ytbl_succ()** to traverse the Ytable from *a1* down to *c1*, and then traverse the Ytable from *b1* back to *d1* using **ytbl_pred()**. Any edges that ended at the transition vertex are deleted during the downward traversal. Any edges that begin at the transition vertex are inserted after the downward traversal. Edges that extend through the vertex do not need to be deleted and reinserted, they can have their order in the Ytable reversed before beginning the upward traversal.

It is necessary to associate with each vertex entered into the Xqueue the edges that emanate from that vertex so that they can be inserted into the Ytable. It is no longer necessary or even possible to associate a bend, start or end "type" as was done in the simple polygon triangulation algorithm. After the vertex has been processed the edges closest to the edges above and below the transition vertex are checked for intersections and if an intersection is found it is entered into the Xqueue.

Another detail facing the programmer during the transition operation is the problem of correctly maintaining the set count arrays associated with the new regions. Although the edges entering the transition vertex were deleted in top to bottom order, the order in which exiting edges are inserted into the Ytable is arbitrary. It depends upon the order in which the exiting edges associated with the transition vertex were inserted into the Ytable. Until the last edge is inserted, the Ytable is in a state that may not correspond to any possible physical configuration and hence it is impossible to update any of the set count arrays until the last edge is inserted. After the last edge is inserted, and the upwards traversal begins, the set count arrays can be updated as each is encountered. We may simply copy the set count array from the previous edge and then increment or decrement the entries that correspond to the sets that the current edge bounds.

The transition algorithm is depicted below:

```
transition(v)
Vertex v;
{
    Edge  h,l,e,next,r,f;
    going_down = TRUE;

    h = ytbl_findabove(v);
    l = ytbl_findbelow(v);
    e = ytbl_succ(h);

    while (e != h){
       r = f = NULL;
       while (r is NULL or f is NULL and e != h){
            if       (e == l) {
               going_down = FALSE;
               insert_exiting_edges(v);
               e = ytbl_pred(l);
            }
            else {
               if       (e is a R edge)   r = e;
               else if (e is an F edge) f = e;
               if (going_down) {
                     next = ytbl_succ(e);
                    ytbl_delete(e);
               }
               else    {
                    next = ytlb_pred(e);
                    fix_set_arrays(e,next);
               }
               e = next;
            }
```

```
        }

        if (r and f are not NULL)
            process the R-F pair
    }
    check_for_intersection(h,ytbl_succ(h));
    check_for_intersection(l,ytbl_pred(l));
}
```

4.4 Handling Edges that Inhabit the Same Line

Edges may exist on the same line and intersect along line segments rather than singles points. This degenerate condition gives rise to dangling edges and polygons with no area. The generalized transition algorithm is not changed appreciably by the steps taken to handle this problem. We let the entries in the Ytable be characterized by a line equation and append a list of the edges that share that line equation. This makes checking for intersections between new neighbors in the Ytable slightly more complicated because we have to ascertain that an intersection of two parent lines is contained by at least one edge lying on each line. Deletions and insertions are also complicated somewhat because we can't delete an entry from the Ytable until all the edges on the line are deleted. Similarly, we have to check to see whether an edge we are inserting already has a parent line in the YTable.

5. Implementation

The generalized algorithm has been implemented in a just under two thousand lines of C. A streamlined implementation suitable for one particular application could no doubt be implemented in much less. Our implementation is augmented to optionally animate its progress and compute certain other information associated with the input. We implemented the Xqueue on top of an implementation of top-down 2-3-4 red-black trees, a balanced tree algorithm. Originally a heap was used but because a heap is only a partially ordered data structure it can not combine identical vertices until they are removed together. Since coincident vertices are the norm rather than the exception in our applications, the balanced tree implementation has performed as well as the heap.

Because our applications have typically involved thousands of vertices and not millions we have gotten away with implementing the Ytable with an array. In practice one can often achieve good performance simply by making sure that the Xqueue is implemented efficiently due to the fact that the Ytable is usually far below worst case capacity at any given time. Intuitively, if the region being swept is roughly square and edges are distributed evenly over the region any vertical line will intersect roughly the square root of the total number of edges. Thus, the Ytable will be filled to the square root of maximum capacity on average. These intuitive ideas are made precise and confirmed by Bentley, Haken and Hon [Bent80]. For the sake of doing things correctly, we plan to implement the Ytable on top of the balanced tree package in the future.

The implementation was coded so that a pointer to the function that evaluates the boolean expression could be passed as an argument to the transition routine. This was a good decision because it allows experimentation with new applications without the duplication of code for the transition routine which is fairly complex.

Our biggest single implementation problem has been tuning the algorithm to allow for floating point errors. Since all decisions in the algorithm are local it really only cares about the relative ordering of values and not about their absolute value. Unfortunately, this is exactly what floating point is not designed to do. The optimal data type to use for vertices and other geometric values would be rational numbers represented with a big integer for numerator and denominator since that would allow exact calculations but this is not possible in our version of C since we are limited to 32 bit integers. Double precision floating point, 64 bit IEEE format, was used for the coefficients of edges in the Ytable but even then a little bit of numerical analysis and experimentation was necessary to prevent edges being inserted into the Ytable incorrectly.

6. Applications

We list here three applications that we have implemented ourselves. We also describe how the algorithm might be employed to solve the hidden surface problem. The key to applying the algorithm is to recognize that a problem can be viewed as a boolean expression. This is not always obvious.

6.1. Arbitrary region clipping

Clearly, the polygon clipping operation is a boolean expression. It is the boolean AND of the clipping window and the rest of the polygons which may be grouped into one set. While we don't advocate implementing clipping an environment to a rectangular window with this algorithm there are plenty other clipping applications that crop up in computer graphics research.

One of our clipping applications is to determine the portion of the boundary of a rectangular volume that is not covered by polygons. We subdivided the model of a building into rectangular volumes and wished to compute the location of openings between subdivisions. Subtracting the polygons inhabiting the plane of the boundary from the rectangular boundary of the volume gave us those polygons explicitly. The subtraction operation is the AND-NOT operation. Conversely, a programmer could place doors and windows in the boundary of a rectangular volume and subtract them away from the boundary to model a room. This type of coplanar CSG operation can be troublesome for CSG algorithms that operate on boundary representation data.

As another example of an unusual clipping application we had an odd shaped region known to be the window in a plane through which two polygons on opposite sides of the plane had to look to "see" each other. Determining that the window was completely covered allowed us to deduce that those two polygons could not see each other. We again used the AND-NOT operation to subtract polygons known to inhabit the plane of the given window from the window. If anything remained after the operation, the polygons could see each other.

6.2 Detecting coincident polygons

It is well known that two objects cannot inhabit the same spot in space. This does not seem to be known by the geometric modelling programs available to many of us. As a consequence, it is possible to model a wall in a building with overlapping polygons. If the two walls have different colors, the result of rendering this object can be disasterous. We have used our program to detect coincident polygons in our models of buildings. The algorithm is run on each set of coplanar polygons in a model. Each polygon is a member of a distinct set. The boolean function is the OR of any two or more polygons. Because our implementation maintains an array with one entry for each set in each element of the Ytable this is not the most efficient method possible. However in this case it works efficiently enough, which in practical terms is the bottom line. An implementation which used linked lists instead of the array or hardwired the boolean function in some way could be used if efficiency became of utmost concern. It is also possible to use something more than a true boolean operation to determine the region of the plane to triangulate. For example a count of the number of polygons covering a region might be used. This is advisable in the cases where the boolean function becomes very large and inefficient to evaluate.

6.3 Transforming a Tiling into a Triangular Subdivision

We define a tiling of a planar region as some decomposition of the region into polygons so that the region is covered at every point by one and only one polygon. A tiling in which edges intersect other edges only at vertices is a planar subdivision. The advantage of a planar subdivision over a tiling becomes clear when the tiling is laid over a curved surface. The tiling of a curved surface may have cracks in it while the planar subdivision avoids this problem. Von Herzen notes this problem in approximating curved surfaces with restricted quadtrees [Von Herzen].

Even if the surface tiled is actually flat, such as the wall of a building, radiosity shading calculations produce a variation in intensity over the surface that can be thought of as a curved surface. Here the cracks will appear as shading discontinuities. Transforming the tiling to a planar subdivision will remove the shading discontinuities.

We made a small modification to our algorithm that allowed it to keep the edges induced by the

tiling and computed the OR of the tiles to get a triangular subdivision. The subdivision allowed proper averaging of radiosity patch values at the vertices and removed the shading discontinuities.

The modification was to treat edges that normally would be ignored, the interior edges, as a Falling and Rising pair of edges rather than as a Constant edge. Since the function passed in as an argument determines whether an edge is a Rising, Falling or Constant edge, this change is fully backward compatible with the implementation as described above.

6.4. The visible surface problem

We have not implemented this application but we describe it because it is such an important problem. The environment could be projected onto the plane. We make each polygon a distinct set. Then compute the OR of these polygons while keeping the interior edges as in the application in section 6.3. The array maintained with each entry in the Ytable tells us which polygons from the input covered each triangle. We can then do a depth test for each of those covering polygons to determine the closest polygon whose surface attributes would be used to render the triangle.

The worst case complexity is $O(n^2 \log n)$ where n is the number of vertices including intersections induced by the projection because the array that must be maintained at each entry in the Ytable is the size of the input. This is another case where a psuedo-boolean function should be used. We want to triangulate everything and keep track of the input polygons that covered each output triangle. The array of the covering polygons is clearly not the best since it will usually have only a few non-zero values and a lot of work will be wasted maintaining the zero values.

However, if a linked list was used to keep track of which sets an edge in the Ytable bounded, it is likely that the average case complexity would rival other rendering algorithms. It would also have the advantage of computing the analytic definition of the visible surface rather than just a point sampling, which allows proper filtering for antialiasing. Furthermore, the rigorous basis of the design would eliminate the glitches produced by all the intersection cases that conventional scanline renderers fail to handle.

7. Conclusion

The applications we have outlined demonstate the utility of a program which can compute boolean combinations of polygons. The plane sweep algorithm provides an efficient method to compute these boolean combinations of polygons. Previous presentations of plane sweep algorithms have stressed theoretical analysis of the algorithm at the expense of completeness. We have stressed robustness and offered a new interpretation of the local geometry of the transition vertex which allows real world input to be processed without extensive special casing.

References

[Bent80]
Statistics on VLSI Designs
Jon L. Bentley, Dorothea Haken, and Robert W. Hon
CMU-CS-80-111, Carnegie-Mellon University, Pittsburgh PA,1980.

[Dobkin]
An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon
David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink
Siggraph 1988 Conference Proceedings, pp. 31-40

[Guibas78]
A Dichromatic Framework For Balanced Trees
Leo J. Guibas and Robert Sedgewick
19th Annual Symposium on Foundations of Computer Science, IEEE, 1978

[Mehlhorn]
Data Structures and Algorithms, Vol 3:
Multi-dimensional Searching and Computational Geometry

Kurt Mehlhorn
Springer -Verlag

[Niev]
Plane-Sweep Algorithms for Intersecting Geometric Figures
J. Nievergelt and F.P. Preperata
Communications of the ACM
Oct. 1982 V 25 No. 10.

[ORourke]
Art Gallery Theorems and Algorithms
Joseph O'Rourke
Oxford Press. 1986

[Ottman]
A Fast Algorithm for the Boolean Masking Problem
Thomas Ottman, Peter Widmayer and Derick Wood
Computer Vision, Graphics and Image Processing, 30 249-268 1985

[Sedgewick]
Algorithms, Second Edition
Robert Sedgwick
Addison Wesley

[Shamos]
Computational Geometry
Michael Shamos and Franco Preperata
Springer-Verlag

[Syzmanski]
Goalie: A space efficient System for VLSI Artwork Analysis,
T. G. Szymanski and C. J. Van Wyk:
IEEE Design and Test, 64-72, June 1985.

[Von Herzen]
Accurate Triangulations of Deformed, Intersecting Surfaces
Brian Von Herzen, Alan H. Barr
Siggraph 1987 Conference Proceedings, pp. 103-110.

[Weiler80]
Polygon Comparison using a Graph Representation
Kevin Weiler
Siggraph 1980 Conference Proceedings, pp. 10-18